LA-UR -82-2928

LA-UR--82-2928

DE83 002064

Conf-830501--1

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

MASTER

TITLE: A METHODOLOGY FOR MINIMIZING MAINTENANCE COSTS

AUTHOR(S): LINDA BRICE AND JOHN CONNELL

SUBMITTED TO PROGRAM CHAIRMAN, NCC '83
Atlantic Richfield Company
515 South Flower Street
Los Angeles, California 90071

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO. 836 R4

A METHODOLOGY FOR MINIMIZING MAINTENANCE COSTS

by LINDA BRICE
and JOHN CONNELL
Los Alamos National Laboratory
Los Alamos, New Mexico

Research conducted in the case study of a large applications system shows that the two primary causes of high maintenance costs are:

1) The frequency of user-requested changes to software; and
2) The psychological complexity of the software.

A "tool kit" is suggested which, when applied to the design of new systems or rewrites, will:

-- produce systems which users are less likely to need changed;

-- contribute to the reduction of psychological complexity of code, making it easier to change when necessary.

The tool kit is easy to use, can be applied to large or small systems in any language on any equipment, and requires no purchase of hardware or software.

Keywords: Applications, Maintenance, Psychological Complexity, User Satisfaction

P. O. Box 1663, MS-P224
Los Alamos, New Mexico 87545
(505) 667-8419
or
4418 Ridgeway Drive
Los Alamos, New Mexico 87544
(505) 662-2236

A METHODOLOGY FOR MINIMIZING MAINTENANCE COSTS

by LINDA BRICE

INTRODUCTION

Maintenance costs escalate when software must be changed. Sometimes there are user-requested changes because the system does not meet the user's needs, and sometimes there are "bugs" because the systems and the individual program modules comprising those systems are not well structured. All changes, whether necessary to fix bugs or desired to improve or add features, are difficult when program code is psychologically complex.

Quantifiable costs associated with software applications include: computer resources utilized by the application; programmer staff time plus computer resource costs expended to maintain the application; and associated user time spent trying to learn and to use the end product.(1) The focus of this paper is on programmer staff time expended to maintain the application. Maintenance will be defined as all changes required to keep a system running according to the user's needs, including:

> Fixes to programs necessitated by coding errors or misunderstanding of user requirements;
>
> changes to programs required due to changes in environment or legal/regulatory changes not under control of the user;
>
> enhancements or optimizations which alter the processing environment, often including minor new features.

Research performed in the case study of a large applications system has shown that the number of changes applied to a

system and psychological complexity (in particular, the misuse of branching instructions) of the code undergoing change both correlate positively with maintenance costs in terms of programmer effort.[2,5] "Psychological complexity," as used here, refers to those elements of programming style which make the resulting software difficult to maintain.

This paper is intended to suggest several aids for the reduction of software maintenance costs. The first suggestion is for the data processing professional to employ certain metrics to estimate the expense of existing software. Software shown to be expensive to maintain may then be subjected to a break-even/payoff analysis for economic justification of a rewrite. When rewrites appear to be economically feasible, care must be taken so that the new system is indeed easier to maintain than the old.

Many data processing shops continue to maintain production systems, despite high maintenance efforts, simply because they work. A method of deciding just when psychological complexity contributes enough to the maintenance costs to be economically unfeasible, would be beneficial. There comes a point in time when, because of psychological complexity due to poor initial program design, or due to many "patches," that rewriting the program (or set of programs) is more economically justifiable than continuing to maintain it.

In order to develop new systems and rewrites of existing ones that will have lower maintenance costs, a methodology is needed for designing with future maintenance in mind. Because psychological complexity is causally related to maintenance costs, the methodology should provide a means

for minimizing such complexity. Since it has been demonstrated that requests from the user for changes correlate significantly with maintenance costs, the methodology should also aim at maximizing user satisfaction with new systems and rewrites in order to reduce future service requests.

WHEN TO REWRITE

One method for deciding when to redesign existing computer applications involves deriving an economic break-even/payoff analysis using a five-step process:

1) Track maintenance costs for a time period and then project future costs, using straight line trend analysis;

2) Measure the complexity of the existing code using a demonstrated metric.[3,4,5] This step does not contribute directly to the break-even/payoff analysis, but it does provide confidence that program complexity contributes to maintenance costs;

3) Estimate cost of rewrites;

4) Estimate costs for the maintenance of the new system after implementation;

5) Prepare a break-even/payoff analysis. In this projection (Figure 1), maintenance costs for the present system are shown as a straight line. Total cost for the proposed system is shown as a broken line with cost to completion of rewrite having a steep slope (it includes cost of maintaining the present system), and cost after

ERT
URE 1
→

completion of rewrite having a gentler slope as the new system will be easier to maintain.

TOOL KIT FOR REWRITE

The life of software systems is traditionally viewed as a cycle or sequence of iterative events. Recently, the life cycle concept has come under fire.[6,7] This paper is not intended to pass judgement on the life cycle concept -- many versions exist, not all without merit. What is proposed here are a few techniques which will hopefully reduce the costliness of maintenance. In order to describe the helpful tools, it is necessary to assume that prior to maintenance, software is written (as opposed to the purchase of a package), and that the development of that software proceeds in some order decreed by management. It is suggested that in order to minimize the number of post-implementation requests from users for changes, users be involved in setting objectives, and that production of output facsimilies and prototyping occur early in the development process.

The assumption will be made that management of DP personnel and management of end users agree to the events which must transpire to get the system up and running. Those events should be scheduled in a visual form (Gantt charts, Figure 2). The events will vary from project to project, but will necessarily include interaction with the user to describe system functions and software development. DP and user management meetings should occur prior to each major milestone for the purpose of schedule review.

SERT
URE 2
→

A METHODOLOGY FOR MINIMIZING MAINTENANCE COSTS
by LINDA BRICE

The major goal is inexpensive maintenance. The tools are recommended on the merits of imposing user interaction during design which leads to fewer user changes, and producing lucid code which results in less effort per change. They are:

System Requirements Definition (SRD)

Tool - Scheduling Guideline;

System Design - Results in System Design Document - (SDD)

Tools - Output Facimilies or Prototypes,
Data Flow Diagrams (DFD's),
Policy Statements,
Data Dictionaries,

Internal Design - Results in Requirements Specification Package (RSP)

Tools - DFD's of Proposed System (from SDD),
Approved Output Formats (from SDD),
Policy Statements (from SDD),
Data Dictionaries (completed from SDD),
Logic-Flow Charts,
Program Abstracts,
Program Design Walkthroughs.

The methods and tools mentioned are not reliant on team makeup or on computer-based tools. None of the tools are original with this paper. What is proposed here is the integrated use of the tools to meet the stated goals.

A METHODOLOGY FOR MINIMIZING MAINTENANCE COSTS
by LINDA BRICE

Systems requirements definition

The Systems Requirement Definition (SRD) will not be covered in-depth in this paper because, except for the schedule, there are no specific tools recommended. The purpose of the SRD is to identify proposed objectives, define the project scope, define the organizational units involved, identify the end-users, identify make or buy approaches and construct a rough schedule and cost/benefit for each alternative.

Cost/benefit analysis has already taken place when the system is a rewrite. It is inherent in the break-even/payoff analysis mentioned under "WHEN TO REWRITE." If the system is an entirely new development, the assumption is that a cost/benefit study would be necessary for a go/no-go decision by management at this point, prior to any actual development effort.

The schedule is not intended to be rigid as to dates. It is intended to identify the tasks to be performed, the parties involved, and the order in which the tasks will be performed. When reviews are held prior to the end of each phase (task), the remainder of the schedule can be reviewed and adjusted for reasonableness.

Gladden warns that "system objectives are more important than system requirements . . . concentrating on objectives can go a long way to prevent a system from 'evolving' into one that the user does not want or need."(7) The life cycle wheel model of system development which concentrates on viewpoints, stresses "...requirements analysis is viewed as a design activity from a user viewpoint. This design is synthesized from various (incomplete, inconsistent) user

scenarios and other expressions of needs. The emphasis is on what functions the system is to perform, and how the system interacts with the users.(8)

The SRD is, then, the project starting point and the place where objectives are defined.

System design

The ultimate degree of user satisfaction with a new system or rewrite is often determined in the early stages of analysis and design. It is recommended that intensive interviews be conducted with the user during this phase. Such interviews should concentrate primarily on net outputs -- the part of the system that will be visible to the user after implementation. Users may have little interest in how data will be massaged to produce these outputs.

System design: document components

Careful users will want to know how the accuracy of the information contained in the net outputs can be guaranteed. A System Design Document (SDD) should, therefore, contain the following elements:

1. A brief description of the framework within which the proposed system will operate, including:

    a) constraints imposed by the operating environment,
    b) required hardware/software configuration,
    c) allowances for future contingencies;

2. Samples of proposed net outputs such as report layouts and screens;

3. Proposed formats for net inputs, showing how data will be captured at original collection points;

4. Visual diagrams of data flows for the present system (either manual or automated) and the proposed system;

5. Policy statements giving a decision method for each procedure shown in the above diagram(s);

6. Rigorous definitions of all data elements shown in the above diagram(s).

System design: tasks

Development of the SDD components need not be undertaken in the order given above. System Design Document development guidelines may specify tasks to be performed in preparing such a document, and the order in which they should be performed.(9) The following is a brief description of each of these tasks.

System design: tasks -- describe system environment

At this stage, the system designer can recognize when the new system will exist in a physical environment which may impose constraints on the design. The task during this phase should involve documenting the nature of that environment and identifying areas which might impose design constraints.

System design: tasks -- describe net outputs

Examples of proposed outputs can be produced rapidly without using actual applications software. The editor on any system can be used to produce a text file which, when copied to the line printer, will produce a facsimile report or screen layout. The main advantage of this approach is that content and format can be changed easily without modifying software. In addition, the facsimile reports provide an immediate focal point for user interviews. Users who tend to be vague about system requirements can often be coaxed into being more specific by discussing information contained in the new outputs. If output formats are approved by the user before system design is begun, the result should be fewer design changes and service requests after implementation.

If an installation has available the necessary tools (i.e., flexible data base systems), it is strongly recommended that a prototype system be brought up at this early stage. "It is now recognized ... that although the customer may state his requirements very firmly at the beginning, his perception of the problem begins to change as he begins to consider how the solution development ... is proceeding."(10) Peters, Gladden, McCracken and Jackson all recommend rapid prototyping to combat wholesale requirements changes.(6,7,10) The remainder of the SDD is charged with demonstrating that the approved sample net outputs can be produced accurately.

System design: tasks -- describe net inputs

If the user is familiar with existing inputs, it is probably not necessary to produce samples. There may be, however,

implications in the above components of the design for new methods of data capture or even entirely new data elements to be captured. In this case, it is important to solicit user approval of new input formats such as data entry screens. The method for providing examples of proposed input formats can be the same as that for output formats -- sample forms produced with a text editor being an obviously simple manner.

Simple design: tasks -- produce data flow diagrams

For a visual representation of the flow of data between functions performed by a system, the use of the Data Flow Diagram (DFD) is highly recommended. DFD's have been explained in Yourdon's structured analysis and design technique, and is described by De Marco.[11] Basically, these diagrams consist of bubbles, arrows, and parallel lines. The bubbles represent a procedure, the parallel lines represent a data store, and the arrows represent the flow of data between the procedures and data stores. The diagrams are leveled as to degree of detail -- the highest (Level 0) contains only one bubble labeled with the system name, and shows only net inputs and outputs to the system (Figure 3). The lowest level diagrams show elementary procedures and data elements (Figure 4). One suggestion for the number of descriptive levels is seven plus or minus two. The rule also applies to the number of bubbles or procedures per level. The diagrams should remain visually digestable as they are the tool for user interviews in this phase.

INSERT FIGURE 3 →

INSERT FIGURE 4 ↗

DFD's demonstrate for the user how net inputs will be transformed into net outputs and, therefore, serve as a

primary check on the accuracy and completeness of the outputs. This technique tends to minimize unnecessary or overly complex procedures and maximize user satisfaction.

Each of the bubbles or procedures shown in the lowest level DFD should have an associated policy statement describing the decision method proposed to perform the procedure. These policy statements should be expressed in structured English or pseudo-code so that they are unambiguous while still intelligible to the user (Figure 5). They should be developed in the interviews with the user so that they are, in fact, the user's policies. Each policy statement should correspond to a bubble on a low-level DFD.

These statements of user policy should eventually become on-line documentation for production source code in the form of prologues (abstracts) for procedure modules. Initially, they serve as a guide to system design; later, they can serve as a maintenance aid.

INSERT FIGURE 5 →

System design: data dictionary

Each of the arrows in all of the levels of the DFD's will have a label. The SDD should include a "dictionary" defining each of these labels. The definition of a data label on a high-level diagram should be in terms of the labels on the next lowest level diagram. At the lowest level, each label should also be defined as to how, when, and where that element will be captured.

If the dictionary is complete and rigorous, it serves as a proof that the user requirements, as expressed in the policy statements, can be satisfied using the data defined therein.

Each definition should correspond to the level of the DFD on which it can be found as the label of a data flow. This also answers the designer's question, "what data do I need, and where can I find it?"

Internal design: requirements specification package

Once the SDD has been approved by the user, "internal" design can begin. Here, internal design will only address those elements necessary to develop low-maintenance software. The Requirements Specification Package (RSP) components will include:

-- Copies of the DFD's which identify program modules;
-- Approved Output (reports);
-- Data Dictionary from the SDD;
-- Policy Statements from the SDD;
-- Logic Flow Diagrams for each module (Chapin Charts);
-- Program Abstracts.

The Data Dictionary may be revised during this phase of the project, and Policy Statements should contribute to the functions listed in the program abstract.

Internal design: chapin charts

It is suggested that Chapin charts,[12] Nassi-Shneiderman Structured Flowcharts,[13] or the Structured Programming Design Method (SPDM)[14] be used to describe logic flow for each program module. The three are similar in philosophy, and any one can be used to bridge the gap between module need (basic requirements) identification and executable code. The document will be referred to here as a Chapin chart.

The lowest level DFD''s in the proposed system section of the SDD represent processes in bubble format. Usually, each of these processes identify a program module, as well as the inputs and outputs. Policy statements in pseudo code or in structured English accompany the DFD's. The combination of inputs, outputs and policy statements form the skeleton of a Chapin chart. If a data base management system is utilized, it will also have been defined in the SDD as "required software configuration" under the Operating Environment. If not, files or specific formats for data transfer mechanisms must be specified prior to construction of Chapin charts.

The Chapin chart is created based upon this cumulative knowledge, sometimes with the addition of special processing algorithms. The reader is referred to the references for in-depth explanations of this logic flow chart. (12,13,14,15) The method, in essence, consists of visually representing a set of program building blocks which allow single entry/exit and strictly limit branching, a practice known to increase psychological intelligibility. The set of program structures includes SEQUENCE, IFTHENELSE, DOWHILE, DOUNTIL, and CASE. When used properly, then the set of combined structures lends itself to a well-structured program guide where arbitrary transfers of control are impossible. Figure 6 is an example of a Chapin chart.

The benefits of the Chapin charts are:

-- Provision of a "GOTO-less" map to be translated directly into a programming language;

-- Provision of a document which graphically depicts logic for the purpose of review (peer review, team walkthrough);

-- Provision of a test bed guide.[15]

It has been noted that Chapin charts are not devices which provide functional hierarchy, interfaces or data flow.[14] The contention here is that there is no necessity for Chapin charts to respond to those needs, as they are met by the DFD. What Chapin charts do well is control flow of executable code within a higher level functional design. This toolkit provides the functional design via DFD's.

INSERT FIGURE 6 →

Internal design: walkthroughs

Approved DFD's showing processes (program modules), inputs, outputs, policy statements, functional hierarchies, interfaces and data flows are available from the SDD phase; program module logic design is graphically represented via Chapin charts. Because of the importance of structuring program code for understandability and readibility effects in the maintenance phase ("good" structure equals psychologically clear code and minimum branching), the Chapin charts should be subjected to a peer review prior to the coding phase. The review should not only insure the structure of the individual modules, but should double check that elements are defined in the data dictionary, that the process will accurately perform what was intended in the higher level diagrams, that the outputs conform to early prototype specifications and that a program abstract is present. The abstract would minimally consist of:

Purpose;
Input (arguments/files/other);
Output (arguments/files/other);

Functions (10 or less);
Local variables;
Subprograms called;
Errors (fatal/non-fatal);
Standards violations.[16]

An example of a program abstract is in Figure 7. The purpose of walkthroughs is improved (low-maintenance) quality of the product. The value of walkthroughs shows up ultimately in the maintenance phase. "The inspection process shifts the discovery and correction of errors and defects from software's operational period to the early design stages. Since the cost for software corrections during operations is many times the cost incurred in detecting problems during design, inspections provide an unusual leveraging of cost/benefit over the entire life cycle of the software."[17] Although a heavy commitment is necessary for the time of team members and moderator participation, other benefits beyond low-maintenance code are accrued, such as "training and exchange of technical information among the programmers and analysts who participate in the walkthrough."[18]

CONCLUSION

Use of this tool kit will not guarantee that the resulting system contains minimal psychological complexity and maximized user satisfaction. It is possible to misuse the tools. The intention of this paper was to explain some of the factors that cause software to be expensive to maintain, and to provide aids that may be useful in designing low maintenance systems.

REFERENCES

1. Brice, L. "Existing Computer Applications -- Maintain or Redesign: How to Decide?." Proceedings of the 1981 Computer Measurement Group International Conference, pp. 20-28.

2. Brice, L., J. Connell, and J. Taylor. "Deriving Metrics for Relating Complexity Measures to Software Maintenance Costs." Proceedings of the 1982 Computer Measurement Group International Conference.

3. Halstead, M. H. Elements of Software Science. Elsevier North-Holland: Elsevier Computer Science Library, New York, New York, 1977.

4. McCabe, T.J. "A Complexity Measure." IEEE Transactions on Software Engineering, (Vol. SE-2, No.1) March 1972, pp. 308-320.

5. Connell, J. and L. Brice. "Complexity Measures Applied to an Applications Case Study." Fourth International Conference on Computer Capacity Management Proceedings, 1982, pp. 121-128.

6. McCracken, D. D., and M. A. Jackson. "Life Cycle Concept Considered Harmful." Software Engineering Notes, Vol. 7, No.2., April 1982, pp. 29-32.

7. Gladden, G. R. "Stop the Life Cycle, I Want To Get Off." Software Engineering Notes, Vol.7, No.2, April 1982, pp. 35-39.

8. Yamamoto, Y., R. V. Morris, C. Hartsough, and E. D. Callender. "The Role of Requirements Analysis in the System Life Cycle." Proceedings of the 1982 National Computer Conference, pp. 381-387.

9. Brice, L., and F. Welch. Manual of Procedures and Standards. Administrative Data Processing Division, Los Alamos National Labortory, Los Alamos, New Mexico, 1982.

10. Peters, L. "Relating Software Requirements and design." ACM Proceedings of the Software Quality and Assurance Workshop, 1978, pp 67-71.

11. DeMarco, T. Structured Analysis and System Specification. Yourdon, Inc., New York, New York, 1978.

12. Chapin, N. "New Format for Flowcharts." Software Practice and Experiences, Vol. 4, No. 4, February 1974, pp. 341-357.

13. Nassi, I., and B. Shneiderman. "Flowchart Techniques for Structured Programming." SIGPLAN Notices of the ACM, Vol. 8, No. 8, August 1973, pp. 12-26.

14. Marca, D. "A Method for Specifying Structured Programs." Software Engineering Notes of the ACM, Vol. 4, No. 3, July 1979, pp. 22-31.

15. Yoder, C. M., and M. L. Schrag. "Nassi-Shneiderman Charts -- An Alternative to Flowcharts for Design." Software Engineering Notes of the ACM, Vol. 3, No. 5, November 1978, pp. 79-86.

16. Control Data Corporation: Final Report of the Aircraft Noise Prediction Program Phase II, (Contract No. NAS1-13983), NASA, Langley Research Center, Hampton, Virginia, July 1978.

17. Werner, F. L. "Software Inspections: Process and Payoffs." Computerworld, April 12, 1982.

18. Yourdon, E. Structured Walkthroughs. Yourdon, Inc., New York, New York, 1978.

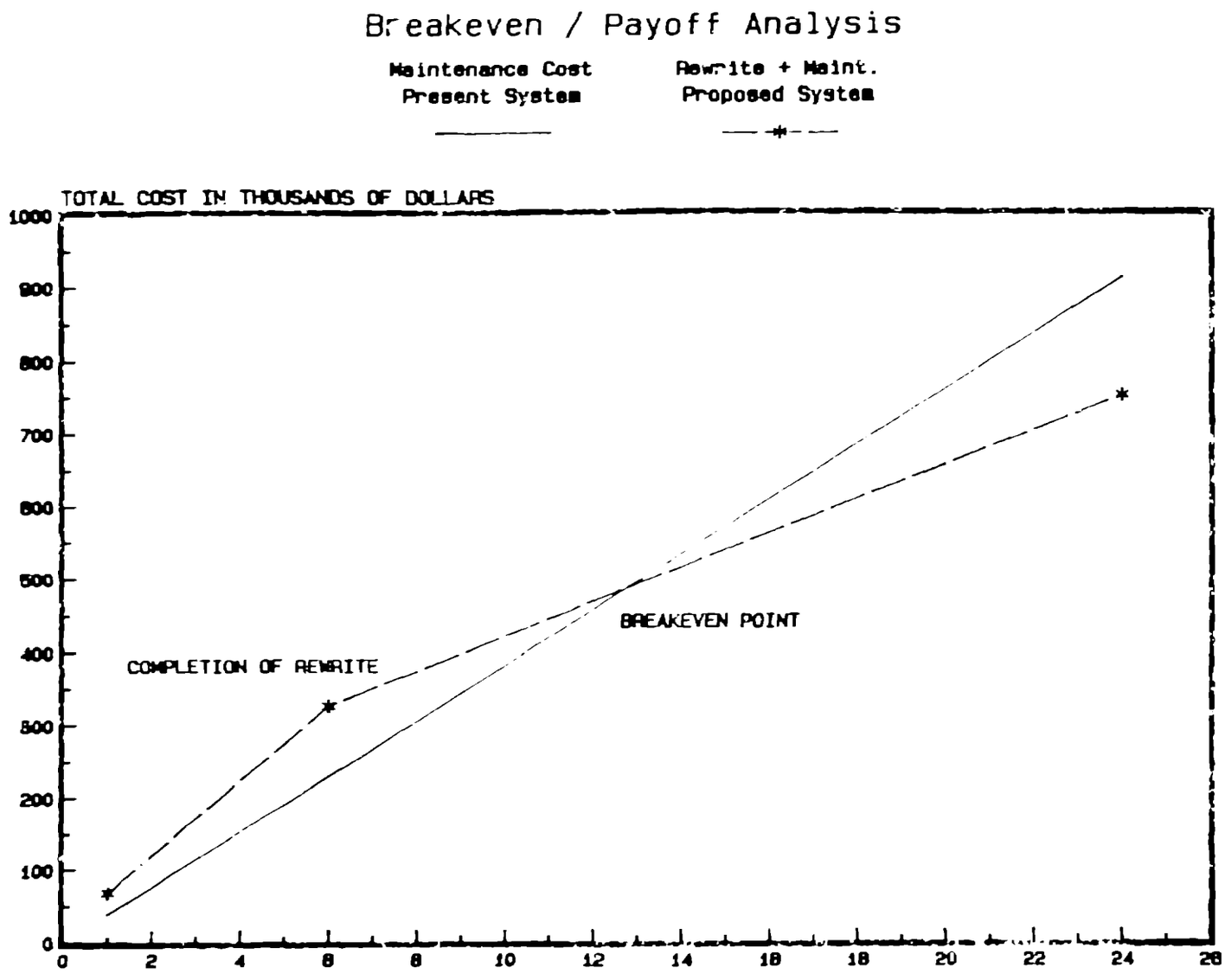

Breakeven / Payoff Analysis
Maintenance Cost
Present System
Rewrite + Maint.
Proposed System
TOTAL COST IN THOUSANDS OF DOLLARS
1000
900
800
700
600
500
400
300
200
100
0
COMPLETION OF REWRITE
BREAKEVEN POINT
0
2
4
6
8
10
12
14
16
18
20
22
24
26
TIME IN MONTHS (Assuming rewrite takes two people six months)

Figure 1.

Widget Division Profit/Loss System Design Schedule

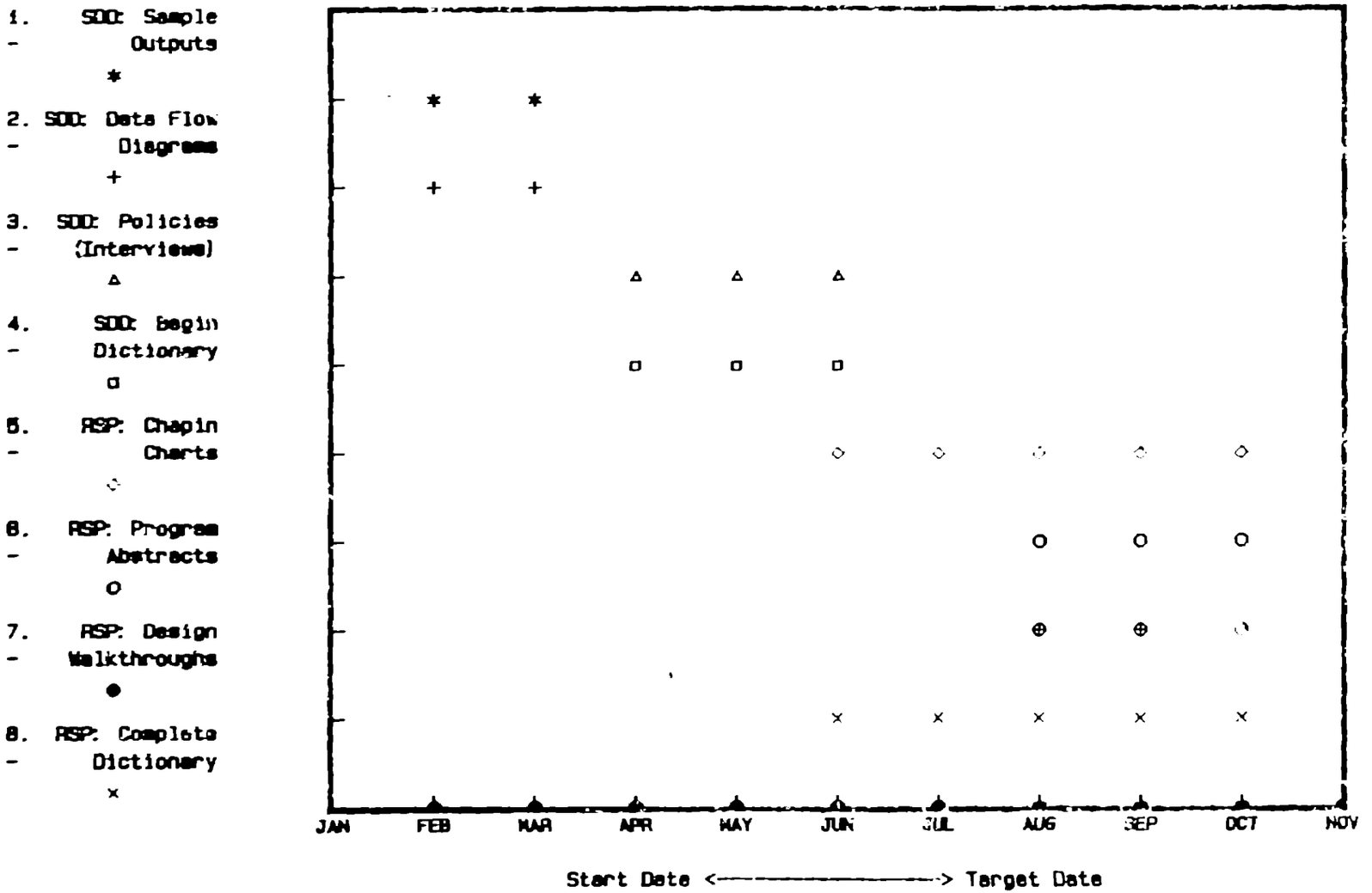
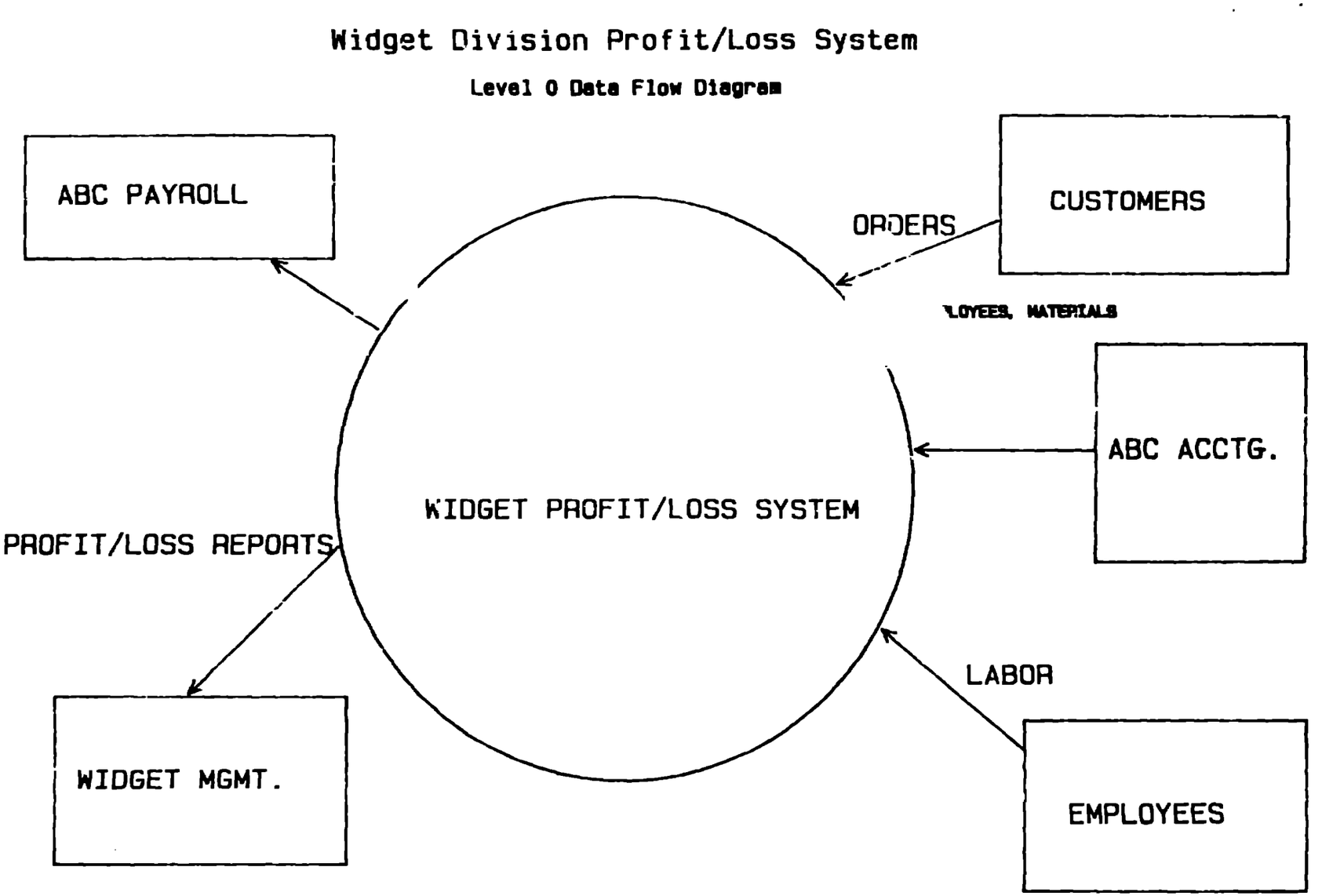


Figure 2.


Widget Division Profit/Loss System
Level 0 Data Flow Diagram
ABC PAYROLL
CUSTOMERS
ORDERS
LOYEES, MATERIALS
ABC ACCTG.
WIDGET PROFIT/LOSS SYSTEM
PROFIT/LOSS REPORTS
LABOR
WIDGET MGMT.
EMPLOYEES

F:GURE 3.

LEVEL 1
WIDGET PROFIT/LOSS SYSTEM DATA FLOW DIAGRAM

FIGURE 4.

(*) REVENUE = LABOR * CHARGE-RATE
COST = (LABOR * SALARY RATE) + OVHD-PER-WORKER

POLICY STATEMENT 1.1
SELECT OVERHEAD COSTS FOR
WIDGET DIVISION

For each ABC Company General Ledger cost record pertaining to Widget Division:

Add salaried employees and hourly employees to employee count;

Add material costed to material-costed sum;

Pass employee count and material-costed sum to 1.2.

For each ABC Company Procurement record pertaining to Widget Division:

Subtract from material-costed sum those purchase orders involving contract labor, resulting in overhead costs.

Pass overhead costs to 1.2 for use in management report.

Divide overhead costs by employee count, resulting in overhead-cost-per-worker.

Update the work-in-progress data base with overhead-cost-per-worker.

FIGURE 5.

CHAPIN CHART FOR POLICY STATEMENT 1.1

SELECT OVERHEAD COSTS FOR WIDGET DIVISION

ENTER

READ FIRST G/L COST RECORD

DO WHILE MORE G/L COST DATA TO PROCESS

IS THIS A WIDGET DIVISION RECORD?

YES

ADD SALARIED-EMPLOYEES TO EMPLOYEE-COUNT

ADD HOURLY-EMPLOYEES TO EMPLOYEE-COUNT

ADD MATERIAL-COSTED TO MATERIAL-COSTED-SUM

PASS EMPLOYEE-COUNT AND MATERIAL-COSTED-SUM TO REPORT ROUTINE

NO

NULL

READ NEXT G/L COST RECORD

READ FIRST PROCUREMENTS RECORD

DO WHILE MORE PROCUREMENT DATA TO PROCESS

IS THIS A WIDGET DIVISION RECORD?

YES

IS IT A CONTRACT-LABOR P.O.?

YES

SUBTRACT CONTRACT-LABOR FROM MATERIAL-COSTED-SUM GIVING OVERHEAD-COSTS

NO

NULL

NO

NULL

READ NEXT PROCUREMENTS RECORD

PASS OVERHEAD-COSTS TO REPORT ROUTINE

DIVIDE OVERHEAD-COSTS BY EMPLOYEE-COUNT GIVING OVERHEAD-COST-PER-WORKER

UPDATE WORK-IN-PROGRESS DATA BASE WITH OVERHEAD-COST-PER-WORKER

EXIT

Figure 6.

PROGRAM ABSTRACT

PROGRAM NUMBER: 56-311

SYSTEM DESIGN NUMBER: 56

DATA FLOW DIAGRAM NUMBER: 1.1

PROGRAM NAME: SELECT OVERHEAD COSTS FOR WIDGET DIVISION

AUTHOR: C. G. POND

PURPOSE: DETERMINE OVERHEAD COSTS FOR WIDGET DIVISION

INPUT:
1. GENERAL LEDGER COST FILE (EXTERNAL FILE NAME = D70129A)
2. GENERAL LEDGER PROCUREMENTS FILE (EXTERNAL FILE NAME = D70101A)

OUTPUT:
1. OVERHEAD COST REPORT
2. UPDATED WORK-IN-PROGRESS DATA BASE

FUNCTIONS:
1. FOR EACH ABC COMPANY GENERAL LEDGER COST FOR THE WIDGET DIVISION, EXTRACT EMPLOYEE COUNTS AND COST OF MATERIAL TO DATE.
2. REDUCE THE MATERIAL COST BY THE AMOUNT OF CONTRACT LABOR.
3. CALL A SUBROUTINE TO PRODUCE AN OVERHEAD COST REPORT, PASSING THE EMPLOYEE-COUNT, THE ORIGINAL MATERIAL-COST, AND THE MATERIAL COST REDUCED BY CONTRACT LABOR.
4. UPDATE THE WORK-IN-PROGRESS DATA BASE WITH OVERHEAD COST PER WORKER. (REDUCED MATERIAL COST DIVIDED BY NUMBER OF EMPLOYEES.)

LOCAL VARIABLES:
MATERIAL-COSTED-SUM
OVERHEAD-COSTS
EMPLOYEE-COUNT
OVERHEAD-COST-PER-WORKER

SUBPROGRAMS CALLED:
1. 56-312 PRODUCE OVERHEAD-COST REPORT

COMPILATION OPTIONS = COBOL5, EL=T, LO.

ERRORS: NONE

STANDARDS VIOLATIONS: NONE

Figure 7.